TICAM REPORT 96-08
February 1996

kScript 2.5 User Manual

Philip T. Keenan

kScript 2.5 User Manual[1]

Philip T. Keenan[2]

February 5, 1996

[1]This research was supported in part by the Department of Energy, the State of Texas Governor's Energy Office, and project grants from the National Science Foundation. The author was also supported in part by an NSF Postdoctoral Fellowship.

[2]Texas Institute for Computational and Applied Mathematics, The University of Texas at Austin, Taylor Hall 2.400, Austin, Texas 78712. http://www.ticam.utexas.edu/users/keenan/kScript.html

Contents

Chapter 1

Introduction

This is the user manual for version 2.5 of *kScript*, a powerful and flexible application scripting language.

1.1 What is *kScript*?

Application scripting languages like Microsoft's Visual Basic, Apple Computer's AppleScript, and the UNIX freeware package TCL, are interpreted programming languages which can be embedded inside other applications, providing users with a high level way to control or *script* the behavior of the application.

Like these other languages, *kScript* can be used as a stand-alone interpreted programming language or embedded inside a larger application. Unlike Visual Basic and AppleScript, *kScript* is not a commercial product and the complete source code for it is available on the World Wide Web at

`http://www.ticam.utexas.edu/users/keenan/kScript.html`

kScript is *not* in the public domain and its use is subject to the terms of the *kScript* copyright notice provided with the distribution and reprinted below in this manual. However, for non-commercial users the terms are not restrictive. Documentation and promotional materials for applications built with *kScript* must cite this *User Manual*, and developers must notify Dr. Keenan (e-mail to `keenan@ticam.utexas.edu`) that they have adopted *kScript* in an application. In particular, the more people let us know they are using *kScript*, the more we will be able to justify allocating resources to supporting, maintaining, and improving *kScript* in the future.

kScript is superficially similar to TCL because both languages evolved over a period of many years from experiments with C interpreters. However, *kScript* was developed independently of TCL and has certain advantages over TCL as a programming language. *kScript* was designed with scientific applications in mind, rather than system level programming. In particular,

- *kScript* is strongly typed. In TCL, everything is a string, and numerical computation is slow and can give incorrect answers (for instance, multiply two large integers.) In *kScript*, numbers are numbers and are stored internally in double precision. Moreover, *kScript* supports a wide and extendible variety of other types of objects including strings, arrays, and files (streams).

- In *kScript* variables are either global in scope or local to a user defined procedure. In TCL, the scope of local variables is not restricted to the body of the function in which they are defined, leading to subtle bug possibilities because the interpretation of a function depends upon the complete calling sequence in which it is executed.

- The *kScript* interpreter accepts a `-safe` command line argument which disables writing to files and access to the UNIX shell. This is intended to make *kScript* safe for use over networks, whether coordinating a parallel computation or providing a Web accessible service. This is an experimental feature, and the author is interested in constructive feedback regarding it.

- *kScript* is implemented in C++ using Object Oriented Programming techniques which make it easy to extend the language. In particular, the `cmdGen` user interface compiler converts a text description of an application into a LaTeXuser manual and into C++ source code, which is linked with the *kScript* library to yield a complete user interface for the application.

For more detailed information about the limitations of TCL, see the WWW page

`http://minsky.med.virginia.edu/sdm7g/LangCrit/Tcl/`

kScript is not a "little language". *kScript* is a complete programming language with *at least* as much expressive power as Fortran 77, which is hardly "little". *kScript* includes comments, typed variables such as numbers and strings, control structures such as looping and branching, and user defined functions and commands. Applications can define additional commands, functions, types and objects which enrich the vocabulary and power of kScript. *kScript* has online help, and if applications are built using `cmdGen`, both the online and hardcopy help are automatically consistent with the actual behavior of the application.

Scripts in `kScript` are similar to UNIX shell scripts; indeed the `shellCmd` command and the `eval` command combine to let *kScript* interact with the UNIX shell in very general ways, thereby enabling powerful extended features like interactive inter-application communication across networks. *kScript* is also similar to MATLAB: it provides an interactive environment for numeric computation. However, the full potential of *kScript* only becomes apparent when applications define additional commands and objects which enrich the language with actions and concepts specific to the application domain.

For example, my `kplot` graphics program extends *kScript* by defining additional commands for plotting lines, triangles, rectangles and polygons. It uses objects to represent color and font palettes, and to control the coordinate system. Moreover, since *kScript* lets users write their own subroutines and functions, it can even be extended with commands to create custom axis tic mark patterns and labels, without any knowledge of the internals of the program or any need to recompile it.

As another example, the Rice Unstructured Flow code (RUF) solves elliptic partial differential equation on general geometries. It extends *kScript* with commands for specifying the computational grid, defining coefficients, and taking one or more time steps. Other scientific computing applications might define objects such as the time step or an error measurement. In this case, *kScript* would allow the user to implement an adaptive time step selection algorithm, and experiment with modifications of it, all without need to understand or even access the source code for the application.

Chapter 2 is a tutorial introduction to scripting with *kScript*. Chapter 3 is the *kScript* Reference Manual. It discusses those commands, functions, types and objects defined by the core *kScript* language and hence available in all applications which support *kScript*. Finally, Chapter 4 describes how programmers can extend the *kScript* language by making variables and procedures within their application known to *kScript*, thereby giving the user detailed control over the execution of the application.

1.2 Copyright and Disclaimer

Chapter 2

Tutorial

2.1 Getting Started with *kScript*

To start with, here is a very simple script.

```
echo {Hello World!}
```

Type this into a file called `hello.k`, and then run the command

```
kScript hello.k
```

You should see the following output:

```
Hello World!
```

You can also use *kScript* interactively. Run

```
kScript
```

and type the `echo` command listed above, or, type

```
include hello.k
```

In either case the "Hello World" message should appear. Notice that when running *kScript* interactively, a `kS>` prompt appears when *kScript* is ready for you to type a command. To exit *kScript*, type

```
quit
```

You can experiment with *kScript* following this example. The `echo` command prints its argument (the text in curly braces) on the screen. The `include` command reads in additional commands from the specified file. The `quit` command exits the program.

There is also a `help` command that explains the on-line help system, which you can use to learn about additional commands and objects. Just type `help` with no arguments to get started.

2.2 A Simple Script

Here is a slightly more interesting script which shows how you can define and use objects such as numeric variables. It will print a simple table of squares and square roots.

```
# a sample script

define command table {Print a simple table.}
  int first
  int last
{
  int i = first
  double square = 0
  double square_root = 0

  echo {%(5r){N} %(5r){N^2} %(10r){sqrt(N)}}
  while i<=last {
    square = i*i
    square_root = sqrt(i)
    echo {%(%5.3g)i %(%5.3g)square %(%10.4f)square_root}
    i = i+1
  }
}

table 1 5
echo {}
table 2 4
```

This script has defined a new command, called `table`, which takes two integer arguments, `first` and `last`. It loops over the specified range, using `echo` to print computed values in a formatted table. Lines beginning with the pound sign (`#`) are comments. Spaces, tabs, and new lines matter only within string literals, which are delimited by curly braces (`{}`). The syntax and semantics are much like in C and C++, except that lines do not end in semi-colons.

All of the commands are explained in more detail in the reference section of this manual. When run, this script produces the following output:

```
N   N^2    sqrt(N)
1     1     1.0000
2     4     1.4142
3     9     1.7321
4    16     2.0000
5    25     2.2361

N   N^2    sqrt(N)
2     4     1.4142
3     9     1.7321
4    16     2.0000
```

In addition to the square root function `sqrt`, *kScript* contains a number of other mathematical functions including exponential and trigonometric functions, as well as uniform and Gaussian

random number generators, as listed in Table 3.2.

2.3 Command Line Arguments

The `kScript` interpreter uses the environment variable `KSCRIPT_INC_DIR` to initialize the `includeDir` string variable. The `include` command looks in this directory if it cannot find the specified file in the current directory. Note that the `directory` variable can be examined and modified to change the current directory.

The `kScript` interpreter starts by including a file named `ks.k`, if one can be found in the current directory or the include directory; then it evaluates any supplied `eval` arguments in the command line, and finally it reads the supplied input script, directing output to the supplied output file.

The `kScript` program accepts the following command line arguments on UNIX platforms:

`-usage` This prints the usage message.

`-transcript wfilename` This causes a transcript of the current *kScript* session to be written to the specified file. This can be useful when debugging complicated script files, or when running codes in interactive mode, as it records all the command you issue and the output that results.

`-eval {commands}` This evaluates the supplied commands after reading 'ks.k' but before reading the main input file.

`-safe` This runs *kScript* in SAFE mode, in which the shell command and file output are disabled.

These options are followed by the name of an input script file, and finally by an output file name; these default to the keyboard and screen if not specified.

2.4 Physical Units for Scientific Applications

One application of *kScript* is to implement systems of physical units in scientific applications. Users can create include files containing definitions for the units of interest to them, after selecting a set of consistent base units in which output will be displayed. For instance, one can define CGS units with a script such as the following:

```
define constant cm 1
define constant m 100*cm
define constant km 10^3*m
define constant gm 1
define constant kg 10^3*gm
define constant s  1
define constant min 60*s
define constant hr 60*min
define constant day 24*hr
```

An application which defined a time step and a domain volume could then be controlled by statements such as

```
set theTimeStep to 4*day+2*hr
set theDomainVolume to (4.5*km)^3
```

which are much easier to read and understand than the input files of many scientific applications!

Similarly one can define a command `print` for output with units, so that

```
print theTimeStep in days
```

prints, for instance, `4.083*day`. Moreover, none of this requires reprogramming or even recompilation of the executable program – users can do this by editing a *kScript* text file.

2.5 Using Array Objects

Here is a simple script which declares a two dimensional array, fills it with values and prints it in 1-D and 2-D formats. As in other languages, one can use use `while` loops or `for` loops for the arithmetic progression of the indices in the iterations. In `kScript`, however, `for` loops can also provide iteration over an enumerated list, either an explicit one or one implied by a keyword collection (described in the next section). Use `while` loops to iterate over numeric ranges that are not simple arithmetic progressions.

```
doubleArray bb = (3,4) base (1,2) row

int i = 0
int j = 0
for i from 1 to 3 do {
  for j from 2 to 5 do {
    bb(i,j) = i-1+(j-2)^2
  }
}

describe bb
echo {Value: '%bb'}
echo {The (1,4) entry is %bb(1,4); the (1,5) one is %bb(1,5)}

string temp = {}
echo temp

for i in { 1 2 3 } do {
  for j in { 2 3 4 5 } do {
    temp = {%temp %bb(i,j)}
  }
  echo temp
  temp = {}
}
```

The output from this script is:

```
Description of doubleArray 'bb'
```

```
2 dimensions: row  major storage
   size      first       last
    3          1          3
    4          2          5

Value: '0 1 4 9 1 2 5 10 2 3 6 11 '
The (1,4) entry is 4; the (1,5) one is 9

 0 1 4 9
 1 2 5 10
 2 3 6 11
```

2.6 String Functions

In *kScript*, commands produce actions but do not return values. Functions are commands that also return values. They can be entered by themselves, like commands, in which case the return value is printed as output, or the return value can be assigned to a variable using the <- syntax variation of the **set** command.

kScript includes several built-in functions for manipulating strings, some of which are illustrated in the following script.

```
function strRest
{Return the rest of the string after the leftmost n characters.}*
  int n
  string input
{
  int len <- strLen input
  string rest <- strRight len-n input
  return rest
}

# Example:

string a = {window frame}

strLeft 4 a
strRest 4 a
strRight 4 a
```

The point is that although *kScript* has built in functions for returning the n leftmost or rightmost characters of a string, it lacks a builtin function for returning the rest of a string after the leftmost n characters. However, this lack is easily remedied, since one can define a function to do precisely this, as shown above.

The output from running the script is:

```
 wind
 ow frame
 rame
```

2.7 Tracing, redirecting output, and generating session transcripts

When scripts become long and start calling lots of user defined commands and functions, it can be helpful to trace their execution, especially when they do not execute as you expected.

The following simple script demonstrates turning on tracing.

```
traceLevel = -1

double x = 3.14
echo {This (%x) appears on the screen.}
x = x-1
echo {This (%x) appears on the screen.}
```

The output is

```
[kS:0] set int [0] traceLevel = '-1'
[kS:0] End   'set'
[kS:0] Begin 'define'
[kS:0] defining double [0] 'x'
[kS:0] set double [0] x = '3.14'
[kS:0] End   'define'
[kS:0] Begin 'echo'
This (3.14) appears on the screen.
[kS:0] End   'echo'
[kS:0] Begin 'set'
[kS:0] set double [0] x = '2.14'
[kS:0] End   'set'
[kS:0] Begin 'echo'
This (2.14) appears on the screen.
[kS:0] End   'echo'
```

It is also possible to redirect output (and input), using stream objects. This can be used, for example, to write different kinds of information to different files, or to control a multi-run simulation in which each run's output is sent to a numbered file.

Moreover, using the `-transcript` command line option, one can get a complete transcript of the input commands and their output, in one file; this is useful for debugging and for preserving the content of interactive sessions.

The following script demonstrates output redirection.

```
stream stdout = outStream
stream myfile = open file my.out for writing

echo {This appears on the screen.}

outStream = myfile
echo {This is written to the file 'my.out'.}

outStream = stdout
echo {This once again appears on the screen.}
```

The output is

```
This appears on the screen.
This once again appears on the screen.
```

In addition, a file called `my.out` was also created and it contains the following line:

```
This is written to the file 'my.out'.
```

If the above script is saved in a file `script.k` and is run using the transcript option, as with the command `kScript -transcript t script.k`, then a transcript file named `t` is also created, which contains the following summary of the run:

```
The kScript version 2 Interpreter: by Philip T. Keenan

# stream.k example

stream stdout = outStream
stream myfile = open file my.out for writing

echo {This appears on the screen.}
This appears on the screen.

outStream = myfile
echo {This is written to the file 'my.out'.}

outStream = stdout
echo {This once again appears on the screen.}
This once again appears on the screen.
```

2.8 Using Keyword Collections

Frequently a scientific application will require the user to select a feature from a finite list of alternatives. For example, a differential equation solver might support several numerical methods such as Euler and Runge-Kutta. Rather than use an integer variable and assign meaningless integer code values to the list of alternatives, one can define a keyword collection and refer to the alternatives with symbolic names. One could do the same thing by defining integer constants, but in *kScript*, collections represent new types, so that the user interface can warn you if you supply an invalid option.

The `category` collection is a built in keyword collection, with literal values such as `Core`, `Stream`, `String`, and `Array`. Each *kScript* command, function or object belongs to a particular category. The online help system can list all members of a given category using the `list` command. For instance,

```
list Strings
```

produces a list of all the string related built-in functions such as `strCmp`, `strLeft`, etc., which are explained in detail in Section 3.5.2. This differs from the command

```
list strings
```

which lists all objects of string type, such as the string variables `directory` and `includeDir`. For a complete list of category names, use the command

```
describe category
```

which can also be used to list the literal values in any keyword collection.

`kScript` has many additional features which you can learn about by browsing the reference section of this manual and trying out your own scripts. Enjoy!

Chapter 3

Reference Manual

For complete and up-to-date lists of all commands, functions, types and objects known to a particular application, run the program and access on-line help. Type

```
help
```

to get started. This reference manual describes the core features of *kScript* found in any kScriptable program. Individual programs can define additional commands, functions, types and objects, as described in Chapter 4, which extend the functionality of *kScript*.

3.1 Commands

kScript reads commands from the keyboard or an input script file and executes them as they are read, rather than storing them for later batch mode execution. This allows the user to have interactive control over scientific applications provided those applications are written in an event-driven style rather than in a batch processing style.

In the reference sections that follow, each command's name is followed by a list of arguments. Most arguments consist of a type name and a descriptive formal argument name, grouped as a pair in angled brackets. These represent required arguments to the command; the actual supplied argument will be parsed according to the syntax rules for the specified type.

Arguments enclosed in square brackets are optional literal strings, typically prepositions. They can be used to create English sentence-like scripts which are easy to read, or they can be omitted with no change in the meaning of the script. Sometimes several alternatives are listed, separated by a vertical bar (|). For example, the syntax of the set command is

```
[set] ⟨nameExpr name⟩ [to|=] ⟨expression expr⟩
```

Both the name `set` and the equals sign are optional, so the five commands

```
set x to 3.14
set x = 3.14
x = 3.14

set x 3.14
x 3.14
```

all assign the same value to a variable named x, but the first three versions are easier for a human reader to understand.

The keywords `optional` and `required` introduce alternative sets of arguments. Each set begins with a string literal which if encountered while parsing the command signals that the remainder of that clause will follow. Multiple cases can be separated by a vertical bar. In the required case, one alternative must be selected; in the optional case, zero or one may be chosen.

The `sequence` keyword introduces an argument pattern which may be repeated multiple times. A sequence argument can be an empty string (`{}`), a curly brace delimited list of one or more instances of the pattern, or, a single instance without the surrounding curly braces.

In *kScript*, a space-delimited sharp or pound symbol (`#`) comments out the rest of the line on which it occurs. Mathematical expressions must be written with no internal spaces. String literals must be enclosed in curly braces, not quote marks. The curly braces can be nested and within them only the percent sign (`%`) is special — all other text is recorded verbatim. In all other contexts, white space (spaces, tabs, line breaks, and so on) serves only to delimit commands and their arguments.

3.2 Functions

Functions are commands that return a value. If they are invoked like a command, the value is printed on the screen, but they can also be invoked using the syntax

```
[set] variableName <- functionName arguments
```

in which case the return value is stored in the indicated variable.

3.3 Types

The formal argument types in command descriptions generally correspond to C++ classes. The actual argument must be in the correct format for the specified type. For an online explanation of the syntax for a particular type T, use the command `describe T`.

Applications can define new C++ types and use them as arguments to commands by defining a text representation and providing an

```
cmdInterpreter& operator>>(cmdInterpreter&, <type>&);
```

function. The `cmdGen` *kScript* user interface generator assumes such an operator is defined for all argument types it encounters.

The `define` and `set` commands can be used to create and modify variables of any *kScript* type. The core `kScript` built in type names are listed below. The command

```
describe types
```

produces this same list on-line.

Types are arranged in a hierarchy. Some entries in this list, like `keywordType`, represent collections of types rather than individual types. The `list` command will list all objects matching a given type, or descended from it in the type hierarchy. For instance, the command `list category` will list all objects (variables) of type category, while the command `list keywordType` will list all objects of any keyword type, including the category type, and the command `list all` will list all symbol names in *kScript*, whether defining objects, commands, or functions.

`all`
: This keyword is the root of the type hierarchy. The command `list all` lists all global symbols regardless of type.

`command`

Commands are defined using the following pattern:

```
define command Name {Description for Online Help}
  Type1 FormalArg1Name Type2 FormalArg2Name ...
{Script to Execute}
```

The task description string must use the curly brace syntax for string expressions. Square brackets in the list of formal arguments can enclose a single optional preposition, which is ignored if encountered when the command is used.

`function`

A function definition is just like a command definition, except that it must use the **return** command in its script to return a value.

`object`

The **describe** command can show the type and value of any object. The **list** command can list all objects, or all variables, or all constants.

`string`

String expressions consist of literal text inside curly braces. The percent sign signals a special construct or the substitution of the value of a variable.

`name`

A name argument can be a string literal in curly braces, which evaluates to a single word with no internal white space, or a single space delimited name.

`numeric`

Arithmetic expressions cannot contain spaces. They can contain literal numbers and numeric variable names, as well as various functions like sin( ) and cos( ), connected by all the usual math operators like +, -, *, /, and either ^ or ** for exponentiation. Use parentheses for grouping to override the usual operator precedences.

`int`

An integer is a whole number

`double`

A double precision number can be written as an integer, or a decimal number, or in scientific notation.

`keywordType`

A keyword type has literal values which may be viewed using the **describe** command. Additional literal values may be created using the **new** command.

`category`

Categories collect groups of related commands, functions and objects to make the **list** on-line help command easier to use.

`stream`

A stream literal has the form

```
open [file] filename for writing
```

Other variations include **reading** and **appending**. Alternatively one can open a string for reading with

```
open string stringVariableName for reading
```

`array`
Any type of array can be defined with any number of dimensions and with any base indices. For instance, the command:

```
doubleArray a = (4,5) base (1,1) col
```

defines a 2-D column major array of double precision numbers with 4 rows and 5 columns each indexed from 1 as in Fortran. For instance, `a(1,1)` is the first element. The command

```
intArray a = (4,5,6) row
```

makes a 3-D row-major array of integers with each component indexed from 0, as in C.

`doubleArray`
An array of double precision numbers.

`intArray`
An array of integers.

`stringArray`
An array of strings.

`assocArray`
An associative array is an array indexed by keyword collection names; the array automatically grows if new literal values are added to the collections indexing it. The constructor syntax follows the pattern:

```
doubleAssocArray a = [collection]
```

Access to entries uses the syntax `a[literal]`.

`doubleAssocArray`
An associative array of double precision numbers.

`intAssocArray`
An associative array of integers.

`stringAssocArray`
An associative array of strings.

3.4 Expressions

Many commands take math or string expressions as arguments. Math expressions can mix numbers, arithmetic and logical operators, and symbolic names. Math operators are listed in Table 3.1.

Logical operators return 1 for true and 0 for false. The `if` command treats 0 as false and nonzero as true. Parentheses override the standard operator precedences. In addition, many standard mathematical functions can be called, as listed in Table 3.2.

Four of these take two arguments (x, y) instead of one. The expression `pow(x,y)` returns x^y, `random(x,y)` returns a uniformly distributed random number in the range $[x, y]$, `normal(x,y)` returns a normal variate with mean x and variance y, and `mod(x,y)` returns x mod y, defined to

Operator	Interpretation
+	addition
−	subtraction
*	multiplication
/	division
^	exponentiation
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!	is not
&	and
|	or

Table 3.1: Math Expression Operators

abs	round	sqrt	exp	log
sin	cos	tan	atan	msec
pow	random	normal	floor	ceiling
mod				

Table 3.2: Mathematical Functions

Construction	Expansion
%%	%
%{	{ without counting toward nesting
%}	} without counting toward nesting
%nX	n repetitions of character X
%\n	newline
%\t	tab
%+	generates no text; this acts as a concatenation operator.
% followed by newline	the newline is suppressed

Table 3.3: String Expression Expansions

be non-negative as long as y is. The underlying random number generator can be re-seeded with the `seed` command. The `msec` function takes no arguments and returns the value of an internal timer, in milliseconds.

String expressions consist of arbitrary text enclosed in curly braces. Inside curly braces, internal white space is not ignored (unless the final right brace is followed by a `*`). Curly braces may be nested. Within the top level of curly braces, one can expand references to other objects' values by preceding their names with a percent sign. Several other combinations are recognized as well, as shown in Table 3.3.

In addition, formatted conversion is possible. For instance, strings can be formatted with left, center, or right justification in a field of a given width. Thus, replacing `%s` with `%(10c)s` will center the text of s in a field 10 characters wide; `l` and `r` give left and right justification. For numbers, use `printf` formatting codes; for instance, `%(%10.5g)x` prints x with 5 significant digits in a 10 wide field.

3.5 Vocabulary

The following subsections list the basic vocabulary of commands, functions and objects built into *kScript* version 2.5. Any program which uses *kScript* as its application scripting language will automatically support them. Such applications typically will define additional vocabulary related to their own subject area; these should be documented in the individual applications' manuals.

3.5.1 Core

`help`
Command: Provides an overview of the on-line help facilities.

`copyright`
Command: Prints the copyright notice and version information.

`list optional {{variable} | {constant}} <keyword* name(kType|category)>`
Command: Alphabetically list all globally defined names of the given type or in the given category. Object type names can be further qualified to list just constant or variable objects. For convenience, the list command recognizes singular and plural forms of type names, and the special forms '`list variables`' and '`list constants`' can be used to list objects of type numeric. Note that category names are usually capitalized. For instance, try '`list strings`' to see the string variables used in the program, or '`list Strings`' to see the commands and functions related to string handling. For a complete list of type names, use '`describe typeNames`', and for a complete list of category names, use '`describe category`'.

`describe <nameExpr name>`
Command: Provides an online description of a command, function, object, or type name. For a command or function this prints the syntax and task description. For an object, it prints the type, value and description. For a type name, it prints the syntax for literals and expressions. For a keyword type, in particular, it lists all currently defined literal values. You can also use '`describe types`', '`describe typeNames`', and '`describe keywordTypes`', for special listings.

`quit`
Command: Stop parsing the current input file here.

`halt`
Command: Exit the program now.

`;`
Command: This command does nothing. It can be used to terminate interactively typed commands that would otherwise look for optional arguments.

`? <nameExpr name>`
Command: This command prints the value of the named object. This is primarily for interactive use – use the echo command for formatted output. For convenience, ? can also print the value of a math expression.

`debug`
Command: Enter the interactive kScript debugger. A subsequent '`quit`' command will return processing to the main script.

`include [file] <nameExpr filename>`
Command: Read commands from a script file.

`directory`
string: The '`current`' directory path, for file names such as those used by the include command.

`includeDir`
string: The directory path used for include files not found in '`directory`'. This variable is initially set to the value of the shell environment variable KSCRIPT_INC_DIR.

`traceLevel`
int: When this number is positive, kScript prints tracing messages for this many levels of nested commands. When negative, tracing is turned on for all commands regardless of how deeply nested. When zero, tracing is off.

`define <keyword* typeName(kType)> optional {{constant}} <nameExpr name> required {{=} <typedExpression expr> | {<-} <typedFunctionExpr fexpr> }`
Command: Define a new variable or constant of some type T. The '`define`' can be omitted if a type name is given. If no type is given, '`double`' is used. The expression syntax must be appropriate for an object of type T. If define is used inside a command or function it creates a local variable whose scope is the remainder of that command or function. If there is a global variable of the same name, the local variable hides the global one and can have a different type. The left arrow syntax can be used to store the return value of a kScript function in the newly defined variable.

`set <lvalue name> required {{=} <typedExpression expr> | {<-} <typedFunctionExpr fexpr> }`
Command: This changes the value of a variable of type T. The word '`set`' is optional. An '`lvalue`' is usually an object name, but with arrays, for instance, it can be an item within the array. The expression syntax must be appropriate for an object of type T. The left arrow syntax can be used to store the value returned by a kScript function.

`beginComment`
Command: Begin an extended comment, which lasts until a matching '`endComment`'.

Extended comments may be nested.

`echo <stringExpr expr>`
Command: Print the value of an object or string expression on the standard output stream, followed by a newline.

`echo- <stringExpr expr>`
Command: Print the value of an object or string expression on the standard output stream.

`error <stringExpr expr>`
Command: Print the value of an object or string expression on the standard error stream.

`shell <stringExpr expr>`
Function: Have the UNIX shell evaluate the expanded string. This function returns the shell's result code. The shell's standard output is redirected to the file '`temp_sys.log`', so any redirection requested within '`expr`' should be followed by a semicolon.

`shellStr <stringExpr expr>`
Function: Have the UNIX shell evaluate the expanded string. This function returns the shell's output as a string.

`eval <stringExpr expr>`
Command: Evaluate the expanded string as if reading commands from an include file.

`if <mathExpr expr> [then] <stringExpr thenCommands> optional {{else} <stringExpr elseCommands> }`
Command: Branching command: if the math expression evaluates to a non-zero value, execute the commands in the string expression following '`then`'; otherwise execute the commands in the string expression following '`else`', if an else-clause is present. Alternatively, one can replace the math expression by a function call by using the syntax '`if <-functionName arguments`'.

`repeat <mathExpr repetitions> [times] <stringExpr commandList>`
Command: Simple Looping command: repeat the commands in the string expression '`repetitions`' times.

`while <mathExpr condition> [do] <stringExpr commandList>`
Command: General Looping command: repeat the commands in the string expression as long as the condition is non-zero.

`alias: <nameExpr newName> [means] <nameExpr oldName>`
Command: Define an abbreviation or alternative name for any existing command, type or object name.

`seed`
int: The random number generator seed. The seed should be a positive integer.

`compare <nameExpr n1> <nameExpr n2>`
Function: This function compares two objects of the same type in a type-specific way and returns a type-specific value. For instance, comparison on numbers returns negative, 0, or positive for less than, equals, or greater than. Comparison on strings also returns on of these three results, based on lexicographical ordering. For most other types, comparison returns 0 for equality and non-zero for differences.

`return <nameExpr symbolName>`
Command: This returns a value from a user defined function. Note that the argument must be the name of an object, rather than a math or string expression.

`continue`
Command: This ends the current iteration of an iterative command such as while, repeat, or for.

`break`
Command: This exits an iterative command such as while, repeat, or for.

`for [each] sequence { <nameExpr varName> } [in] <stringExpr sourceList> [do] <stringExpr commandList>`
Command: Iteration command: There are three variations of the '`for`' command. The most general version reads a sequence of already defined variable names, then reads values for them from the source list. If there are N names and N*S items in the source list, then the commandList is executed S times, once for each set of values. For instance:

```
for each {i s} in {1 {hi} 2 {there}} do {...}
```

In the keyword version, instead of a sequence of names, a keyword collection type name and a variable name are given, and the source list is omitted. The keyword version executes the commandList once for each literal value in the keyword collection; the literal values are assigned to the variable name in the same order in which they are listed by the describe command, namely newest to oldest definitions. The iteration variable is created local to the scope of the command list. For instance:

```
for each category c do {...}
```

The third version is for arithmetic progressions. The syntax is

```
for <name> from <mathExpr first> [to] <mathExpr last>
  optional {{step} <mathExpr step>}
```

The named variable must be a previously defined numeric object, which is set to be the values from first to last inclusive, stepping by 1 if no other step is specified.

3.5.2 Strings

`strLen <stringExpr str>`
Function: This function returns the length of the string.

`strCmp <stringExpr str1> <stringExpr str2>`
Function: This function returns negative, zero, or positive values as str1 precedes, equals, or or follows str2 alphabetically. Unlike the generic '`compare`' command, this applies only to strings, so literal strings are allowed as well as string variables. In particular, one can compare keyword variables and literals this way, as strings.

`strPosn <stringExpr sourceString> <char marker>`
Function: This function returns the position of the marker character in the string, or 0 if it is not found. The character can be surrounded by curly braces, to represent special characters like spaces and braces.

`strRevPosn <stringExpr sourceString> <char marker>`
Function: This function returns the position of the marker character in the string, starting from the end, or 0 if it is not found.

`strLeft <mathExpr n> <stringExpr sourceString>`
Function: This function returns the leftmost n characters of a string.

`strRight <mathExpr n> <stringExpr sourceString>`
Function: This function returns the rightmost n characters of a string.

`strMid <mathExpr first> <mathExpr last> [from] <stringExpr sourceString>`
Function: This function returns the indicated substring (an inclusive range of characters, counted from 1)

`today`
Function: This function returns the current date as a string.

3.5.3 Streams

`read <stream s> sequence { <nameExpr symbolName> }`
Function: This reads one or more values from the stream into the indicated variables. For each symbol name, the command determines the type of the variable and reads a literal or expression of that type. If the end of the file is encountered unexpectedly, the function prints an error message and returns -1. Otherwise, it returns 0. Note: one can only read into plain variable names, not into math expressions, string expressions, or array expressions.

`close <stream f>`
Command: This closes the indicated file.

`streamWidth <stream f> <mathExpr width>`
Function: This sets a new width and returns the old one. It can only be used with application defined output streams.

3.5.4 Arrays

`dimensions <nameExpr arrayName>`
Function: This returns the number of dimensions in the array.

`firstIndex <nameExpr arrayName> <mathExpr d>`
Function: This returns the first valid index in dimension d of the array.

`lastIndex <nameExpr arrayName> <mathExpr d>`
Function: This returns the last valid index in dimension d of the array.

`resize <nameExpr arrayName> <stringExpr dimList> optional {{base} <stringExpr baseList> } required {{row} | {col}}`
Command: This allows an array to be resized using the same syntax as an array definition.

`inputArray <nameExpr arrayName>`
Command: Input literal values (no math expressions) into a one dimensional int or double array using any of 6 syntax options, including:

```
N : d1 ... dN
N { d1 ... dN }
d1 ... dN end
{ d1 ... dN }
N from filename
from filename
```

where "filename" is expected to hold N double precision numbers (only one array per file). When N is not known, this reads to the end of the file. This routine resizes the array, but preserves it's base index.

`sort <nameExpr inputArray> <nameExpr outputArray> optional {{perm} <nameExpr outputPermutation> }`

Command: Sort a one dimensional array. The outputPermutation is stored in an intArray, such that inputArray[outputPermutation[i]] == outputArray[i].

`arithMean <nameExpr inputArray>`

Function: Returns the arithmetic mean of a one dimensional array of numbers.

`variance <nameExpr inputArray>`

Function: Returns the variance of a one dimensional array of numbers.

3.5.5 Bit

`bitShift <intMathExpr i> [by] <intMathExpr n>`

Function: Shift bits of an integer, left for $n > 0$, right for $n < 0$.

`bitAnd <intMathExpr i1> <intMathExpr i2>`

Function: Bitwise AND of two integers.

`bitOr <intMathExpr i1> <intMathExpr i2>`

Function: Bitwise OR of two integers.

`bitNot <intMathExpr i>`

Function: Bitwise NOT of an integer.

`bitXor <intMathExpr i1> <intMathExpr i2>`

Function: Bitwise XOR of two integers.

3.5.6 Utility

`type <nameExpr n>`

Function: This function returns the type of the given symbol, as a literal value from the keyword collection '`kType`' (as listed by the command '`describe types`'). If n has type '`name`', it's value is examined and if it is a symbol name, that symbol's type is returned instead.

`new <keyword* typeName(kType)> <nameExpr name> <nameExpr parent> <stringExpr documentation>`

Command: This defines a new literal value in the specified keyword collection type. The parent literal (from the same collection) is optional. Use '`keywordType`' as the type name to create a new keyword collection type.

`isDef <name n>`
Function: Returns true if the name is defined in the current scope.

`isLit <name n> [of] <keyword* typeName(kType)>`
Function: Returns true if the name is a literal of the given keyword collection.

`isConst <name n>`
Function: Returns true if the named variable is actually a read-only constant.

3.5.7 Misc

`internalStats`
Command: Prints information about the hash table.

`monitorMemory`
int: Set this to a non-zero value to check for memory leaks.

`maxSyntaxErrors`
int: The maximum number of syntax errors allowed before the program halts. Set this to -1 if you do not ever want the program to stop because of syntax errors.

Chapter 4

Extending the kScript Programming Language

This chapter contains advanced material and may be skipped by beginning users.

4.1 Extensions at the scripting level

4.1.1 Overloading command names by type

kScript is designed to be very easy to extend. To begin with, at the scripting level, user defined commands are very flexible. The `eval` command allows evaluating data (a string) as code (commands), much like in LISP and related languages. This means users can design new control structures directly in *kScript*, without access to the C++ implementation.

For instance, one can set up type based overloading of function names (as in C++) by scripting functions along the lines of the following example:

```
double d = 3.14 ; string s = {hi there}

command show_double {} double nd { echo {this is a double '%nd'} }

command show_string {} string ns { echo {this is a string '%ns'} }

command show {a demo OOP command} name n {
  string t <- type n
  eval {show_%t %n}
}

show s ; show d
```

This script prints the following:

```
this is a string 'hi there'
this is a double '3.14'
```

4.1.2 Defining new types using string lists

`kScript` allows advanced users to define new data types at the scripting level. The following simple example illustrates a `point` type consisting of two numeric fields (x and y coordinates).

```
double px = 0
double py = 0

command getPoint {Put coords. into px, py}
  string p
{
  stream f = open string {%p} for reading
  name k = {}
  int state <- read f {k px py}
  if <- strCmp k {point}
    {echo {not a point: '%k'}}
}

command printPoint {Print a point}
  name n
{
  getPoint n
  echo {Point %n: x = %px, y = %py.}
}

string p1 {point 0.5 0.7}
string p2 {point 0.66 0.9}

printPoint p1
printPoint p2
```

The output is:

```
Point p1: x = 0.5, y = 0.7.
Point p2: x = 0.66, y = 0.9.
```

4.1.3 Defining new types using associative arrays

Here is another way to define a new type, this time using associative arrays rather than string lists. The following input script again defines a point class and produces the same output as in the previous example.

```
new keywordType pointField {A point class stores 2 doubles: x and y.}
new pointField x {The x coordinate}
new pointField y {The y coordinate}

function point {Evaluate return string to define a new point}
  name n
  double xv double yv
{
  string ret = {
    doubleAssocArray %n = [pointField]
    %n[x] = %xv
    %n[y] = %yv
```

```
  }
  return ret
}

command printPoint {Print a point}
  name n
{
  eval {double xv = %n[x]}
  eval {double yv = %n[y]}
  echo {Point %n: x = %xv, y = %yv.}
}

string temp <- point p1 0.5 0.7
eval temp
temp <- point p2 0.66 0.9
eval temp

printPoint p1
printPoint p2
```

4.2 Extensions at the C++ source code level

Programmers can extend *kScript* by connecting it to a scientific application using `cmdGen`. The `cmdGen` program is a "user interface compiler". It provides an easy way to build *kScript* user interfaces to application programs. A `cmdGen` script defines the commands, objects and type names that will be used in the interface. The `cmdGen` program converts this script to the C++ source code needed to implement the specified application interface, and also creates a LaTeX manual documenting the interface. The program understands very general command and function descriptions, in which there can be optional arguments or repeated sequences of arguments.

kScript is a context sensitive language. Unlike traditional context-free languages, argument parsing is under the control of individual commands. This makes left context sensitivity easy to implement; moreover, the command interpreter class can handle one object look-ahead, making limited right context sensitivity also straightforward to use. `cmdGen` takes the place of a traditional parser generator (like `yacc` or `bison`), allowing programmers to quickly and easily define and implement new commands, complete with automatically generated on-line help.

The user manual for `cmdGen` contains further information on how programmers can connect applications to *kScript*.

Bibliography

[1] Keenan, P. T., *cmdGen 2.5 User Manual*, Texas Institute for Computational and Applied Mathematics, University of Texas at Austin, February 1996.

[2] Keenan, P. T., *TUF 2.5 User Manual: The Texas Unstructured Flow Code*, Texas Institute for Computational and Applied Mathematics, University of Texas at Austin, February 1996.

[3] Keenan, P. T., *RUF 1.0 User Manual: The Rice Unstructured Flow Code*, Dept. of Computational and Applied Mathematics Tech. Report #94–30, Rice University, 1994.

[4] Keenan, P. T., *kScript 1.0 User Manual*, Dept. of Computational and Applied Mathematics Tech. Report #95–02, Rice University, 1995.

[5] Keenan, P. T., *C++ and FORTRAN Timing Comparisons*, Dept. of Computational and Applied Mathematics Tech. Report #93–03, Rice University, 1993.